L•A•C•I•R

Laboratory for Automation, Communication and Information Systems Research

Department of Computer Science

DCS-197-1R

Kevo - a prototype-based object-oriented language based on concatenation and module operations

ANTERO TAIVALSAARI

Technical Report - LACIR 92-02
University of Victoria

Kevo – a prototype-based object-oriented language based on concatenation and module operations

Antero Taivalsaari
University of Victoria, B.C., Canada[1]
antero@csr.uvic.ca

June 30, 1992

Kevo is a portable, interactive object-oriented language based on a simple and concrete prototype-based object model. Unlike many previous prototype-based systems, which use *delegation* to achieve incremental modification (inheritance), Kevo is based on *concatenation*: linear composition of object interfaces. The concatenation model provides a concrete metaphor for object-oriented programming: objects may be seen as directories such as those in conventional file systems, and the self-reference denotes the "current working directory" among these objects. Flexible module operations are introduced to achieve incremental modification in various ways. In addition to its unique object model, Kevo features multitasking, several operating system-like features such as flexible input/output redirection, and powerful reflective facilities like runtime decompilation, debugging, and execution tracing. The Kevo system shows that the essence of object-oriented programming can be captured using only a very small number of tangible and intuitive ideas.

1. Introduction

A recently presented alternative for conventional class-based object-oriented programming is prototype inheritance, or object inheritance [Bor81, LaP85, Bor86, Lie86, LTP86, Ste87, UnS87, SLU88, LaL89]. As opposed to class-based systems, in prototype-based systems there are no classes; new object types are created directly by constructing concrete, full-fledged objects known as prototypes or exemplars. A prototype may be thought of as a standard example instance [Bor86] which represents the default behavior for some concept [Lie86]. In general, a prototype is

> *the first or primary of anything; the original of which another is a copy, imitation, representation, or derivation, or is required to conform; a pattern, model, standard, exemplar, archetype* [OXF33 p.1512].

Prototype-based object-oriented programming is often equated with the notion of *delegation*. Many papers, including [Lie86, Ste87, AkT88, Hal88, Mer88], in fact imply that there exists a dichotomy between inheritance (≈ class inheritance) and delegation (≈ prototype inheritance). In this paper this dichotomy will be refuted, though. Besides that delegation can well be seen as underlying class inheritance [Ste87], it is quite possible to build a prototype-based language which does not incorporate delegation but which nonetheless provides the same basic expressive power.

---

[1]This paper was written while the author was visiting the University of Victoria.
Permanent address: University of Jyväskylä, Finland. E-mail: tsaari@jyu.fi

This paper describes a new programming language – *Kevo* – which can be seen as an alternative for delegation-based prototypical object-oriented languages such as *Self* [UnS87, SLU88, CUL89, ChU91]. Kevo is a portable, interactive, prototype-based object-oriented language which features encapsulation, both early and late binding, multitasking, flexible input/output redirection, and powerful reflective facilities. The main contribution of Kevo is however its extremely simple and concrete object model. The Kevo system features neither conventional inheritance nor delegation, but the incremental modification of objects is achieved using concatenation: linear composition of object interfaces. Flexible module operations are introduced to achieve object composition and modification in various ways.

The remainder of this paper is organized as follows. First, Section 2 discusses the theoretical background underlying the concatenation-based approach. Section 3 describes the fundamental principles of the Kevo language: module operations, the basic programming style, and system organization. Section 4 introduces other interesting features of Kevo such as multitasking and reflection. Section 5 describes the status of the current Kevo implementation, and Section 6 compares Kevo to related work. Finally, Section 7 contains some concluding remarks.

2. Theoretical background: delegation and concatenation

> "Broadly speaking, the history of software development
> is the history of ever-later binding time."
> -- *Encyclopedia of Computer Science*

One of the central features of object-oriented programming is *inheritance*: the ability to receive properties from previously defined objects. Inheritance is still quite a controversial notion, and it has different meanings and uses in different contexts. For some researchers, inheritance is strictly a conceptual *specialization*, or *subtyping* mechanism. For others, it is a practical mechanism for reusing existing implementations (see [Ame87, RaL89, Ame90, CHC90, PaS91]). Here a more general view is adopted, and inheritance is regarded as a general incremental modification mechanism whose primary purpose is to promote software development in an incremental manner rather than by editing and/or copying existing code. In order for this incremental modification to operate correctly, a mechanism called *late binding* is needed. Late binding allows the determination of references to the properties of an object to be delayed until the program is actually run. By virtue of late binding, an object's behavior may vary dynamically depending on the context from which its properties become invoked. This context is usually determined by so-called *self-reference*. In general, the following definition captures the essence of inheritance in object-oriented programming (adapted from [Coo89, CoP89]):

> **Inheritance** is *an incremental modification mechanism in the presence of a late-bound self-reference.*

If we simplify the analysis of inheritance to object interfaces (identifier name spaces), then inheritance can be viewed generally as some kind of a *combination operation* [Coo89, CoP89, WeZ88, BrC90]

$$R = P \oplus \Delta R$$

on record structures $P = [...]$ and $\Delta R = [...]$, where R denotes a newly defined object or class, P denotes the properties of an existing object or class, ΔR denotes the incrementally added new properties to differentiate R from P, and $\oplus$ denotes an operation to somehow combine ΔR with the properties of P.

Within computer memory there are two basic ways to express direct relationships between things: *references* or *contiguity*. Based on this observation, it is possible to distinguish two elementary strategies of implementing inheritance: *delegation* and *concatenation*. Delegation is a form of inheritance in which the interface combination is implemented by *sharing* the interface(s) of the parent(s) – in other words, using references. Concatenation, on the other hand, achieves the same effect by *copying* the interface of parents into the interface of the descendant – as a result of this, the descendant's interface will be contiguous. As an example of delegation and concatenation, consider the following definitions. Let us assume that in our system we have the following kind of object *Parent*, which has two variables and one operation (method). Since we are now interested only in interface management, the actual values of variables and implementations of methods have been omitted:

Parent :- [VAR pv_1; VAR pv_2; METHOD pm_1].

Suppose that we now want to define another object *Child* which inherits the properties of *Parent*, and adds one new variable and one method. If we accomplish this using delegation, the interface of the descendant will be as follows:

Child :- [PARENT p :- Parent; VAR cv_1; METHOD cm_1].

On the other hand, using concatenation the interface of the resulting object will be contiguous, as illustrated below:

Child :- [VAR pv_1; VAR pv_2; METHOD pm_1; VAR cv_1; METHOD cm_1].

In both cases the basic effect is the same: the descendant will be able to respond to messages corresponding to the inherited identifiers in addition to the incrementally added ones. However, the way how message sending must be handled in these approaches is different. In delegation, messages which are not accepted by an object must be *delegated* (forwarded) to parents. In concatenation, in turn, the interface of every object is self-contained, and thus no forwarding is needed. Since the term delegation implies *shared responsibility* for the completion of a task, it would be unreasonable to use that term to denote also the latter kind of inheritance. Observe that both forms of inheritance necessitate the presence of a late-bound self-reference, because otherwise the inherited operations

might not be able to refer to the incrementally added properties. Delegation and concatenation have been characterized in Figure 1 and Table 1.

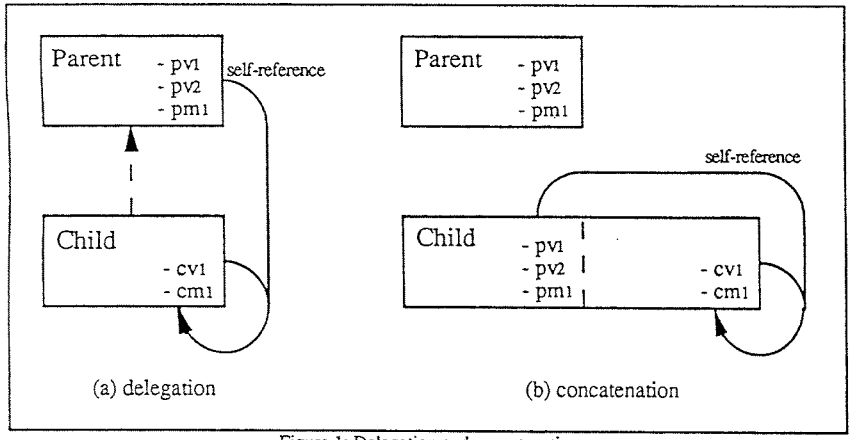

Figure 1: Delegation and concatenation

Delegation and concatenation both provide certain advantages over each other. For instance, owing to sharing of interfaces, delegation enables flexible propagation of change; any change to the properties of parents will automatically be reflected to descendants. The concatenation approach, on the other hand, allows identifier names of parents and children to be modified independently of each other, which may be useful in many situations. Owing to shared interfaces, delegation is obviously more space-efficient, but the contiguous interfaces in concatenation are easier to optimize, and thus the concatenation approach can be regarded as more time-efficient.

Inheritance strategy	Delegation	Concatenation
Record combination strategy	sharing/references	copying/contiguity
Interfaces (name spaces)	dependent	self-contained
Inheritance hierarchies	preserved	flattened

Table 1: Delegation versus concatenation

Note that the question of delegation versus concatenation is independent of whether an object-oriented system is based on classes or prototypes; delegation and concatenation can underlie equally both models. The main difference is that in class-based systems delegation or concatenation takes place between classes, whereas in prototype-based systems the same occurs between individual objects. In fact, although the term *concatenation* is not widely used in the literature of object-oriented programming, there are good historical reasons for its use. The concept of *prefixing* in Simula [DMN68], which later evolved into the modern concept of inheritance, was initially defined in terms of *textual concatenation of program blocks* [CoP89] (see [DDH72 pp.202-204, NyD78]). Even in the current implementations of Simula and its descendant Beta [KMM83] the interfaces (virtual operation search tables)

of classes and their subclasses are self-contained and independent of each other [MaM88, Mag91]. This is different from for example Smalltalk-80, in which subclasses physically share the interface of their superclasses [MaM88]. On the other hand, as shown by Stein [Ste87], the way how inheritance between classes ("metaclass inheritance") is achieved in Smalltalk is analogous to the delegation approach adopted in prototype-based languages such as Self. In both approaches unresolved messages will always be forwarded to separate constructs; in Smalltalk this separate construct is a superclass, and in Self it is a parent object.

In general, delegation and concatenation are two distinct strategies for implementing inheritance, both of which can equally underlie class-based and prototype-based systems. Simula and its descendant Beta are examples of languages which implement *class inheritance by concatenation*. Smalltalk, on the other hand, implements *class inheritance by delegation*. Self is the best example of a language that implements *prototype inheritance by delegation* (see Figure 2). This paper shows that prototype inheritance by concatenation also forms a feasible basis for language implementations. In fact, such languages are not merely feasible, but prototype inheritance by concatenation seems to result in conceptually cleaner and more economical systems; such systems also possess features which are not inherently available in other models. These issues will be examined in the following sections.

	Class-based systems	Prototype-based systems
Delegation	Smalltalk	Self
Concatenation	Simula, C++	Kevo

Figure 2: Inheritance schemes in different systems

3. Principles

"Do not share your objects beyond necessity."
-- Occam's razor reformulated

After the theoretical introduction to concatenation, let us now focus on the more practical principles underlying the Kevo system. In particular, it will be explained how it is possible to achieve incremental modification in the absence of any conventional form of inheritance/delegation. Special module operations will be introduced to allow individual

objects as well as groups of objects to be tailored flexibly at runtime. Furthermore, it will be illustrated how a concatenation-based system can provide a logical and intuitive structuring of objects, in spite of the seeming lack of any hierarchical organization in such a system. Finally, the concatenation-based approach will be compared to prototype inheritance by delegation.

3.1. Module operations

> "There are six kinds of change: generation, destruction, increase, diminution, alteration, change of place."
> -- Aristotle, *Categories §14*

The concatenation model is basically a subset of prototype inheritance with delegation. There are only one kind of objects, no classes, and no notion of instantiation. The effect of instantiation – the ability to create a finite number of similar objects – is achieved by *cloning* (copying) existing objects instead. As in prototype inheritance by delegation, all the objects are basically *individually modifiable*. This means that objects can be modified and extended on the individual basis, as opposed to class-based object-oriented systems in which typically only groupwise (class-level) modification of objects is possible. The way how new kinds of objects can be derived from previous ones is however considerably different in delegation and concatenation.

Delegation-based systems such as Self put a strong emphasis on *sharing*. Programs are developed by building upon existing components, using existing objects as parents, thus explicitly sharing their code and data. This seems appropriate since the meaning of the term delegation is *shared* responsibility for the completion of a task. Heavy emphasis on sharing has its drawbacks, though. Since the delegation model enforces sharing to be used even when the conceptual modeling situation would not necessarily demand it, the programmer's attention is often unnecessarily shifted from conceptual design level to implementation-level aspects. For instance, in Self objects are typically broken up in two pieces – traits and prototypes – so as to make the propagation of dynamic changes as easy as possible [UCC91]. This solution is clearly an implementation-level arrangement – not something necessitated by conceptual modeling. In general, the use of delegation inherently creates strong implementation dependencies between program components, and these dependencies – even though they can be useful at times – decrease the individuality of objects, hamper conceptual design, and make the implementation of things such as concurrency and distribution more complicated.

In Kevo, a different approach has been adopted. We believe that the use of sharing should be limited to situations where it is really demanded by the conceptual model, and otherwise sharing should be used only as an internal optimization technique of which the programmer need not be aware. Basically, each object should be as individual as possible. Propagation of dynamic changes should obviously also be supported, but this should be an explicit operation, rather than something which happens implicitly if an object happens to have descendants.

To promote flexible and sharing-independent composition of objects, Kevo features special *module operations*. These operations can be seen as an extension of the module operation formalism introduced recently by Bracha [BrL92, Bra92]. In that formalism, the term *module* designates anything that supports an *interface*: a specification of a set of services. Modules are seen as records, and module operations determine the various ways of how these records can be combined and modified. Basically, there is nothing new in such operations; similar formalisms have been introduced earlier in different contexts [Fra71, Wan89, HaP91]. Bracha however shows that by adding the notion of the late-bound self-reference, the module operations capture the essence of inheritance in the same manner as the semantics of inheritance were explained by Cook [Coo89, CoP89]. In an applicative (rather than imperative) framework, where each module operation can be viewed as producing a new module rather than modifying existing modules themselves, *all the module operations can naturally be seen as forms of inheritance* [BrL92]. The module formalism is independent of the issues such as whether a system is based on classes or prototypes, delegation or concatenation, or reference versus copy semantics in general, and in this sense the module formalism captures what has been the objective also in the present author's thesis [Tai91b].

The Kevo system incorporates most of Bracha's module operations in an imperative language framework. Some of the operations have different names, and some of them have been implemented in a different fashion to make them more more practical. Kevo also extends the module operations by adding the notion of *propagation*, which allows the module operations to be applied not only to single abstractions, but to groups of abstractions as well. Since Kevo is a prototype-based system, the term abstraction here refers to individual objects rather than classes. In general, module operations in Kevo can be categorized in two different groups:

1) object-specific, and
2) group-specific.

An object-specific module operation has an effect only on a single object in the system, whereas group-specific module operations will be applied individually to each object in a group of objects. Group-specific module operations have been designated using '*' in the end of the operation name (as in Kleene closure).

Kevo's module operations for object extension have been gathered in Table 2. The basic syntax for using each of these operations is *'object OPERATION property'*, where *'object'* is any legal expression to get an identity of an object (to which the module operation will be targeted), *OPERATION* is the name of the module operation, and *'property'* is the name of the property which will be affected by the operation. The slight exceptions to this syntax are the operations *ADDS* and *ADDS**, which allow multiple properties to be added at the same time (see next subsection).

Operation	Behavior
ADDS	Add properties to an existing object
*ADDS**	Add properties to all the objects in a group of objects
REMOVE	Remove a property from an object
*REMOVE**	Remove a property from each object in a group of objects
RENAME	Rename a property in an object
*RENAME**	Rename a property in each object in a group of objects

Table 2: Module operations for object extension

Unlike Bracha's notation, which incorporates two separate keywords for the overriding form of property addition (*OVERRIDE*), and non-overriding property addition (*MERGE*), Kevo uses only a single operation *ADDS* (and the corresponding group-specific operation *ADDS**) for both of these forms of property addition. This reduces the need for source code editing if the same piece of source code is loaded into different contexts. In general, one of three different qualifiers can be supplied by the user to specify how the addition of properties is to take place. This specification can be performed via the user interface, rather than having to hard-wire the qualifiers within the source code.

override	overlapping properties will be overridden (the default)
merge	overlapping properties will be ignored
redefine	overlapping properties will be redefined

In addition to the above mentioned object extension module operations, Kevo also features module operations to control the visibility of properties in objects (encapsulation). The module operations for encapsulation have been gathered in Table 3. Besides using these operations, the visibility of properties can be specified also in a more conventional manner when new kinds of objects are being declared.

Operation	Behavior
HIDE	Make a certain property in an object private (encapsulated)
*HIDE**	Make a certain property in each object in a group private
SHOW	Make a certain property in an object public (non-encapsulated)
*SHOW**	Make a certain property in each object in a group public

Table 3: Module operations for encapsulation

Note in Table 3 that again both object-specific and group-specific forms of module operations have been provided. What is exactly the purpose for making this distinction? The basic reason is quite simple. Remember that every object in Kevo is truly individual, and thus modifications to objects will normally have effect only on single objects. For instance, if one adds a new operation to an object, other copies (≈ instances) of that object will remain unaffected. However, in many cases there may occur a need to make changes to larger groups of objects. Conventional interactive object-oriented systems support such groupwise modifications relatively well, provided that the properties to be added are operations rather than variables. In Self, for instance, runtime method additions to parent objects will be automatically reflected to descendants owing to the use of delegation. The similar effect occurs also in class-based systems such as Smalltalk when methods are added

to classes. However, when it comes to the addition of (instance) variables, things become much more complicated, since complicated structural changes may be required in instances. Self does not support such propagation of variable slots at all, and this has been mentioned as one of the major stumbling block in the design of Self and almost all the other current object-oriented languages [SLU88].

Kevo does not suffer from the above mentioned limitation. Group-specific module operations allow flexible propagation of different module operations regardless of whether the affected properties are operations or variables. Kevo's solution is also more general in the sense that it allows the programmer to explicitly decide whether or not a specific object modification should be object-specific or group-specific, whereas in delegation a modification to an object will inevitably have an effect on all the descendants of that object.

Kevo provides three special qualifiers which allow the objects affected by a group-specific module operation to be specified in more detail. This specification is based on the similarity of object interfaces. In general, a group-specific module operation can have effect on:

1) *clone family*	All those objects which have been cloned from the same prototype, and which still have the precisely same interface.
2) *descendant family*	All those objects which have been cloned from the same prototype, and which still have the same underlying interface.
3) *derivation family*	All those objects which have been cloned from the same prototype.

The group (1) is the most restrictive and can be regarded as the same as the "instances of one class" in class inheritance. Group (2) is an extension of (1) in which also "instances of subclasses" will be affected, provided that no "inherited" properties have been removed or renamed or undergone any other major changes in "subclasses". Group (3) is the same as (2) but without the above mentioned restrictions. In addition to the above mentioned group specifications, we have also planned to support a less restricted form of propagation in which the user can explicitly select the target objects using the graphical user interface. Note that the above mentioned three families of objects can also be used as a basis for dynamic type checking, if the user so desires. At the present time type checking has not however been a major concern in the design of Kevo.

3.2. Incremental modification in the absence of delegation

"Repetition is the only form of permanence that nature can achieve."
-- George Santayana

The module operations described in the previous subsection provide a very flexible and comprehensive platform for constructing and modifying object-oriented programs. In fact, the previous subsection may have given an overly complex first impression in the sense that in everyday programming only a subset of these operations is actually needed. This

subset is very small indeed, and it comprises the heart of concatenation-based programming.

How do we capture the essence of inheritance – incremental modification – without any conventional form of inheritance or delegation? In an imperative language framework, the module operations have inherently effects only on existing objects, and produce no new abstractions as a result. However, by producing a new copy of an object and by applying module operations to the copy rather than the original object, this problem can be avoided, and the resulting new object is an incremental modification of the original one. In general, the following definition captures the essence of concatenation-based programming:

> Concatenation = Copying + Application of module operations,
>
> or more generally:
>
> Inheritance ≈ Copying + Modification.

That is, in concatenation new kinds of objects are formed by copying existing structures, and by subsequently modifying the resulting new objects using module operations. In the minimal case this modification means simply the addition of new properties using the *ADDS* operation. Late binding of self-reference is obviously required so as to guarantee that the modifications take place in an incremental way and the previously defined operations can adapt to these modifications without recompilation. Recall that both of the required operations – copying (cloning) and incremental addition of properties to individual objects – exist as elementary operations even in delegation-based prototypical systems, and in this sense the concatenation scheme can be seen as a simplification of delegation. In a way, delegation is a redundant mechanism even in delegation-based systems (!), albeit the elimination of delegation *per se* would cause some expressive power to be lost.

Let us consider an example. Suppose we wanted to build a simple tennis video game in the spirit of the first-generation video games. In such a game two paddles were used to control two simple bats to hit a small bouncing ball. For generality, it is assumed that the game allows more balls to be added to the game as the game proceeds. In this kind of a game, there are clearly two kinds of basic abstractions: bats and balls. Since more than one copy of both bats and balls is needed, both abstractions can be seen as clonable, graphical objects, which share some common features with each other. Therefore, let us first define a simple object GraphicThing, which defines the general features of graphic things. The object is defined by first cloning the object Object so as to "inherit" the basic printing and creation operations; after that, variables *x* and *y*, and an operation *draw* are then added to the newly created object by applying the module operation *ADDS* to the newly created object. The keyword *'REF'* is actually a shorthand for a certain kind of *ADDS* operation; this will be explained in the next subsection.

```
REF GraphicThing
Object.clone -> GraphicThing
```



```
GraphicThing ADDS   \ Apply a module operation to object GraphicThing
        VAR x       \ Define a local variable x
        200 -> x    \ with an initial value 200

        VAR y       \ Define a local variable y
        50 -> y     \ with an initial value 50

        METHOD draw     ... ;     \ Define an operation draw
ENDADDS;
```

Objects Ball and Bat are constructed in a similar manner by first cloning the object GraphicThing so as to "inherit" its properties, and then adding the appropriate new variables and operations to these new objects using module operations.

```
REF Ball
GraphicThing.clone -> Ball

Ball ADDS
        VAR xStep     \ x coordinate increment
        VAR yStep     \ y coordinate increment

        METHOD bounce       ... increment ball location ... self.draw ... ;
        METHOD serve        ... initialize ball location ... self.draw ... ;
ENDADDS;


REF Bat
GraphicThing.clone -> Bat

Bat ADDS
        METHOD moveUp       ... move bat upwards ... ;
        METHOD moveDown     ... move bat downwards ... ;
ENDADDS;
```

Bear in mind that all the objects in a concatenation-based systems are prototypical, and every single object can serve as a basis for further modifications. No conceptual differences between the first instance of a new object and its copies exist. For instance, in the above example new balls and bats could be created simply by cloning any of the previously existing copies of those objects (in this case there is only copy of each).

The concatenation approach and the example given above are remarkably simple and straightforward. Each object is essentially a linear composition of the cloned and newly added properties, so the object structure is intuitive and easy to understand. Conceptually, each object has its own interface which is totally independent of other objects in the system. The operations of objects are also conceptually totally independent of other objects, although the actual implementation of may however wish to optimize this intuitively rather memory-consuming strategy using virtual or lazy copying techniques [MBK86]. Such optimizations have been heavily used in the current Kevo implementation without sacrificing the individuality of objects. The variables of objects are also typically

independent of other objects. This however depends on what kind of cloning operation has been used (if needed, cloning operations can be redefined for each object). Kevo also provides a keyword *SHAREDVAR* which can be used within the *ADDS* and *ADDS** module operations to declare variables which are always shared between an object and its derivation family regardless of cloning. Such variables can be seen as the equivalent of Smalltalk's class variables.

Is the suggested scheme equivalent to delegation? Basically it is; copying and module operations capture the effect of delegation, group-specific module operations achieve the propagation of dynamic changes, and lazy copying techniques optimize the memory consumption. One important issue which seems to be still missing is multiple inheritance, but actually the implementation of that is relatively straightforward when one realizes that multiple inheritance is simply a generalization of module (interface) combination [Tai91b, Bra92]. The current version of Kevo features a special *multicopy* operation which copies and joins several objects together so as to create a single linear object which can then be incrementally modified in the normal manner. The resulting form of multiple inheritance is *ordered* [CUC91], which is less flexible than *prioritizable* multiple inheritance in Self, but quite common in other object-oriented systems [Moo86, BKK86, BDG88]. These and other differences between delegation and concatenation will be summarized in Section 3.4.

3.3. Organizing programs without classes or delegation

> "This structure of concepts is formally called a hierarchy and since ancient times it has been a basic structure for all Western knowledge."
> -- Robert M. Pirsig, *Zen and the Art of Motorcycle Maintenance*

One of the problems in most class-based object-oriented systems of today is the lack of proper system organization when large applications are being built. Namely, classes and possible global variables typically reside in a single flat, globally accessible name space, which does not provide much organization when the number of classes in the system grows large [BoO87, UCC91]. Some systems, such as Smalltalk-80, divide the name space into a two-level hierarchy in order to organize the name space into smaller, more easily digestible class *categories*, but even these solutions are inadequate in many cases.

Owing to the lack of classes, prototype-based object-oriented systems seem intuitively less organized than class-based ones [Mer88]. Subsequent experience has however shown that this need not be the case [UCC91]. Namely, in prototype-based systems with delegation it is quite possible to define objects which are composed solely of parent or component references to other objects, and which may thus serve as *name space objects* to organize the other objects in the system into more manageable hierarchical categories. This possibility has been utilized effectively in Self which features an object called *lobby* from which all the other objects in the system can be accessed – either directly or via the "subtrees" of lobby [UCC91]. These subtrees organize the Self system conveniently so that objects with a similar purpose can be kept close to each other and clearly apart from other objects in the

system. Since arbitrary levels can be created for object organization, this solution is much more general than that in Smalltalk-80 [UCC91].

In designing the Kevo system organization, we have striven to achieve the same level of generality and flexibility as in Self. Similarly as *lobby* in Self, Kevo features an object called *Root* which divides the system into clearly distinct subtrees for different kinds of objects. The basic difference between Self and Kevo is however that Kevo does not support delegation, and hence the basic system organization can be seen as a pure part-whole (component) hierarchy. Each object forms a separate node in this hierarchy, and the interfaces of these nodes are conceptually fully independent of other objects in the system. In a way, this organization is analogous to conventional hierarchical file systems such as that in Unix or on a Macintosh; each object is a directory/folder, and every single object creation can be viewed as the equivalent of adding a new directory to the hierarchy (either directly as a named object, or indirectly as an anonymous member of some collection object). Individual properties in this hierarchy can be accessed/invoked analogously to Self using combined message sends to form paths from *Root* to the desired property (e.g. Prototypes.Stack.clone). Encapsulation may obviously restrict such access by making certain properties inaccessible.

Note that the idea of using the part-whole hierarchy as the basic skeleton for system organization has some inherent limitations. In particular, since a part-whole hierarchy describes only component relationships, it does not tell anything about the possible commonalities and similarities between objects in the system. The objects which have been cloned from the same prototype can well be located in various places in the part-whole hierarchy, and this makes type checking and comparison of objects inherently quite difficult. In order to overcome this limitation, the Kevo system automatically maintains another object hierarchy in addition to the user-managed part-whole hierarchy. This other hierarchy is based on clone families and their relationships with respect to each other.

In general, the Kevo system supports two kinds of object hierarchies. The *organization hierarchy* (part-whole hierarchy) serves as the basic skeleton for system organization and is managed by the user. The *behavior hierarchy* (clone family hierarchy) captures the commonalities between objects (clone families) and is managed automatically by the system on the basis of module operations. These two hierarchies are orthogonal to each other, but both of them describe precisely the same set of objects. Normally the user deals primarily with the organization hierarchy, but the behavior hierarchy – which can be thought of as being superimposed on the organization hierarchy – has several important uses. In particular, since the behavior hierarchy implicitly keeps track of all the similar objects in the system and their behavioral relationships, it can serve as a tool for type checking and maneuvering in the organization hierarchy. As explained in Subsection 3.1, this hierarchy also serves as a basis for possible propagation of module operations. Note that the relationships between clone families do not imply anything about possible sharing/duplication of properties between/within the clone families; this is strictly an implementation issue.

The two hierarchies in Kevo have been illustrated in Figure 3. To avoid extra complexity, some clone families and some of the references from the behavior hierarchy to objects in the organization hierarchy have been left out.

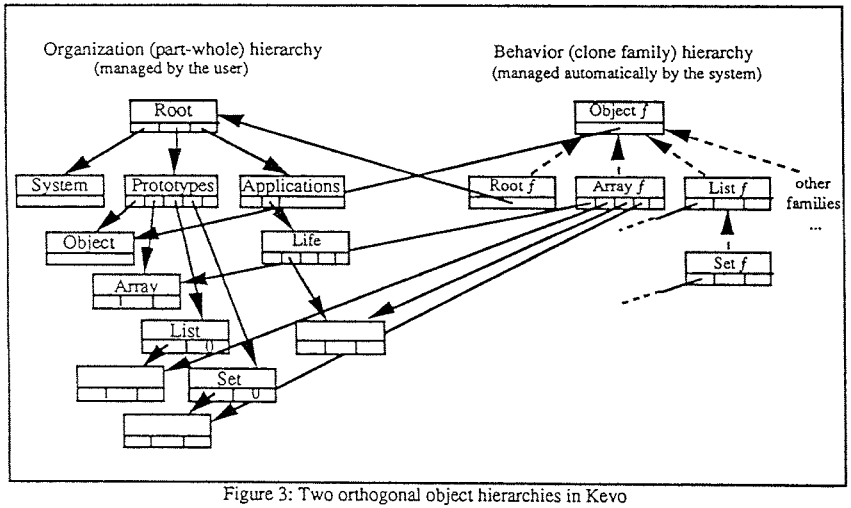

Figure 3: Two orthogonal object hierarchies in Kevo

One curious offspring of Kevo's system organization is the file system-like module operation *'CD'* which allows the user to "go into" an object. When a *CD* operation is applied to an object, the self-reference is changed to that object, thus making that object the "current working directory" in the system. Subsequently all the commands will be implicitly evaluated in that object's context. This feature is particularly interesting in the presence of multitasking (see Section 4), since it allows the user to easily inspect and manipulate concurrently running task objects. In general, this arrangement turns the self-reference into something much more concrete than in conventional object-oriented systems.

> Aside: the keyword *'REF'* used in the examples earlier in Subsection 3.2 is simply a shorthand for a special kind of module operation. For instance, the phrase *'REF x'* is the equivalent of:
>
> ```
> self ADDS
>     SHAREDVAR x
> ENDADDS;
> ```
>
> This operation adds a shared variable slot *x* to the object designated by the self-reference. If the self-reference were *Root*, the new slot would thus be added to the root of the system.

We are currently in the process of building an object browser for the Kevo system based on the above described system organization. In the browser, objects are illustrated as folders such as those in the Macintosh Finder™, and the user can navigate in the system simply by opening and closing these "folders" in the normal manner. On the basis of any given

object, the user can request the system to display the corresponding clone family as an object with the members of the clone family as the properties of that object. Via the clone family object, it is then possible to inspect and manipulate the other members of the clone family, or to move to parent or descendant clone families. Object copying and all the module operations will obviously be supported by the browser so as to allow flexible incremental modification at runtime.

The characteristic feature of concatenation-based programming is the true individuality of objects. Unlike in Self, in which the behavior of an object is typically a sum of a potentially large number of separate structures, in concatenation every object is self-sufficient and not a mere fragment in a tangled traits/prototype hierarchy. The idea of self-sufficiency will also be the characteristic feature of Kevo's object browser. A single "folder" always embodies all the properties of a particular object. When the user wants to use an object, or to learn how it works, no maneuvering is needed but every essential aspect of that object can be found in the same place. This approach makes programming very concrete, and also helps reasoning about programs e.g. by alleviating the yo-yo problem [TGP89, Mey90].

3.4. Summary: concatenation versus delegation

The relative merits of prototype-based and class-based programming have been discussed extensively elsewhere [Bor86, UnS87, Ste87, Weg87, SLU88, Weg90], so the suggested model will not be compared to class inheritance here. Instead, let us now briefly summarize the main advantages of the suggested concatenation-based object model over prototype inheritance by delegation.

- The model is conceptually very simple, and easy to teach about.
- Objects are self-sufficient, so in order to understand the behavior of an individual object one does not have to learn a potentially large hierarchy of structures.
- The model allows all kinds of new properties, including local variables, to be added (propagated) to groups of objects easily, whereas in delegation typically only operations and shared variables can be added.
- Linear, self-sufficient object interfaces hide the implementation decisions about sharing/duplication, and allow for easier optimization of message dispatching in general.
- Since sharing is not required at the conceptual level for achieving incremental modification, the model is more suitable for the implementation of concurrent and distributed programming.

On the negative side, the following issues should be mentioned:

- The concatenation model does not support unanticipated sharing of object *state* as comprehensively as prototype inheritance by delegation.
- Since the concatenation model eliminates parent slots, dynamic inheritance – the ability to change the parents of an object at runtime – is more complicated than in delegation.

- For the same reason, the model does not support prioritization of multiple inheritance as flexibly as Self.

Due to the linear nature of objects, the concatenation model is inherently less flexible than delegation. In Self, for instance, dynamic inheritance – the ability for objects to change their inheritance patterns at runtime [SLU88, UCC91, CUC91] – can be achieved simply by changing the contents of parent slots. Since there are no parent slots in Kevo, the similar effect would necessitate explicit removal and addition of properties in an object's interface. Similarly, to achieve the effect of parent prioritization, one would have to manipulate the order of properties in an object's interface. Although direct manipulation of object interfaces is basically possible, it does not sound very elegant, and therefore the current Kevo implementation does not support these features. However, it can be argued that dynamic inheritance and prioritization of multiple inheritance – although they do have some interesting advantages [UCC91, Tai92] – are still quite controversial language features, and it is by no means certain that a good object-oriented system should support them. Even among the developers of Self there seems to be some disagreement on whether these features should really be allowed [CUC91]. Their arguments for avoiding these features are basically two-fold; first, the interaction of these mechanisms with other language features is complicated and difficult to explain; second, these mechanisms hamper the optimization of message dispatching and can thus slow down the system.

The other basic limitation of concatenation, the inability to fully share object state in an unanticipated manner, arises from the elimination of delegation. Primitive objects, such as integers and characters, are logically individual, immutable objects (or to be more precise: values [Mac82]), but for efficiency they are typically stored in mutable variable slots using containment rather than references. The lowest-level object creation facility that Kevo provides is shallow copying, but as obvious, shallow copying of objects which contain such primitive objects inevitably results in duplication of these primitive objects, and hence sharing of primitive state does not seem possible. Note that the same limitation exists in delegation-based systems, too, but there the sharing of primitive state can be achieved using the delegation mechanism instead.

Basically there are two possible ways to address this problem. If we want the concatenation model to be fully equivalent to prototype inheritance by delegation, we must require the implementation of concatenation to be based on *full reference semantics*. This implies that even the smallest primitive objects should be represented as separate, possibly mutable objects. Using this solution, shallow copying becomes a sharing mechanism, which is capable of supporting shared primitive state. The first experimental implementation of Kevo used this approach [Tai91a], but unfortunately this lead to quite an inefficient, impractical implementation. In general, full reference semantics seemed to be too strong a requirement, considering that sharing of primitive state is an exceptional situation rather than the norm: in most cases the desired result of object copying is that the primitive values are *not* shared. Therefore, the current implementation uses a more conventional solution, and stores primitive objects directly using containment. Sharing of primitive state, when needed, can be achieved by placing these primitive variables in a separate mediating object, which can

then be referred to as a variable from other objects, and accessed by sending messages in the normal manner.

In general, we prefer the linear, self-contained nature of concatenated objects, because it provides a concrete and intuitive metaphor for programming: an object is simply a collection of its properties, nothing more and nothing less. Each object describes its own behavior, and the only basic relationships between objects are the component relationships between objects and their variables and methods. The approach implicitly hides most of the implementation decisions pertaining to sharing/duplication, and thus lends itself more readily to various internal optimizations as well as concurrent and distributed programming. It can be claimed that the concatenation approach is inherently better suited to the implementation of prototype-based systems than the delegation approach, considering that the whole idea of prototype-based programming is based on the conception that objects should be as concrete and tangible as possible.

4. Features

In addition to its unique object model, the Kevo system has other interesting features, as summarized below.

Interactiveness, extensibility, and portability. Kevo is designed to be an *interactive* system. All the definitions can be given interactively via the user interface, or alternatively loaded from some other device using inbuilt input/output redirection. Reasonable execution speed is gained using indirect threaded code [Dew75] as object code. All the input lines are compiled to such code; depending on the context, the compiled code either becomes a permanent part of the system, or is executed right after compilation and then immediately disposed of. Memory for all the definitions is allocated dynamically, and everything in the system is basically dynamically resizable or redefinable.

Like many other interactive systems, one of the key characteristics of Kevo is *extensibility*. The language kernel provides only a set of primitives, and everything else – object definition facilities, command shell, control structures etc. – is then defined in terms of these primitives and other definitions. As a result, e.g. new control structures can be added to the system flexibly if needed. Another main goal in building Kevo has been *portability*. At the moment, two versions of the system exist – one for Macintosh and one for Unix – and except for the Macintosh user interface these systems share basically the same source code.

Encapsulation; early and late binding. Like most object-oriented systems, Kevo supports *encapsulation*: the ability to exclude some properties of objects from their external interfaces so as to gain better representation independence. In Kevo this is achieved by declaring properties *hidden* while they are being defined, or by using the module operations described in Subsection 3.1. Hidden properties are visible during message sending only when messages are sent via the self-reference – that is, from "inside" an object. Note that since all the objects are self-sufficient, Kevo does not make any C++-like distinction

between *private* and *protected* properties (see [Sny87]); within an object, all the properties are assumed to be accessible.

Kevo provides very flexible means for specifying *binding* of properties. Unlike Self, which supports only late binding, or Smalltalk-80 [GoR83, GoR89], which uses early binding for variables and late binding for operations, Kevo does not make any implicit assumptions about binding but allows the user to explicitly decide what form of binding to use when he accesses the properties. All the properties – regardless of whether they are operations or variables – are basically candidates for late binding (in C++ terminology: *virtual*). The actual form of binding can be specified using the notations illustrated in Table 4 below. In the table, *'x'* is the name of the property to be bound, and *'obj'* denotes some existing object in the system.

Notation	Meaning
x	early binding via self-reference
self.x	late binding via self-reference
resend.x	late binding via self-reference (starting from currently executing operation)
obj.x	late binding to another object

Table 4: Forms of binding in Kevo

To our knowledge, the only two other object-oriented languages which follow the same kind of convention in specifying binding are *Neon* [Duf84, Sch86] (later known as *Yerk*) and *Actor* [Duf86, Whi90]. Note that Kevo (deliberately) does not allow early-bound references to properties which do not reside in the object's own interface. We believe this is a reasonable limitation, because early-bound references to other objects would impose strong implementation dependencies between objects and thus hamper the individual and incremental modifiability of objects.

Multitasking. Kevo supports multitasking: the ability to create a finite number of concurrently executing tasks running in the same program. Tasks in Kevo are full-fledged objects, which can be set to execute any given piece of code, activated, and suspended simply by sending messages to them. Since tasks are objects, they can obviously be stored to other structures, cloned, passed as parameters, or otherwise treated like other objects in the system. By default, each task is set to execute the system command shell, and this feature gives the user the ability to effortlessly create a basically unlimited number of command shells. Unix-like definition of background tasks is also supported from any command shell. Each task has its own execution stacks, textbuffers, files and compilation state variables, and depending on the type of the task, each task can have zero, one, or multiple windows. All the tasks however operate in the same object hierarchy using their self-reference as the "current working directory", so the memory occupied by individual tasks is very small (typically less than two kilobytes).

The current Kevo virtual machine supports two modes of multitasking: preemptive and cooperative. Cooperative multitasking has slightly better overall performance, but unless explicit task switching commands are added to computation-intensive places in code, other tasks can become blocked. All Kevo's I/O primitives however contain a call to the task

switcher, so that as long as the task does I/O, it is automatically guaranteed to cooperate with other tasks. In the preemptive multitasking mode (the default), the system takes care of task switching automatically without user intervention. This mode allows tasks to have priorities which determine how much execution time each task receives. In the Macintosh implementation these priorities, as well as the activation and suspension of tasks, can be controlled flexibly from the user interface. Except for the user interface connections, Kevo's multitasking has been implemented in a fully portable and device-independent fashion.

At the moment Kevo has no built-in object-level synchronization primitives. The system however provides means for specifying critical regions, and a small library of basic objects such as semaphores and shared queues, streams, mailboxes, etc. which automatically take care of mutual exclusion when their properties are being invoked. Certain timing services are also provided by the kernel to control the execution of tasks. Owing to the reflective nature of the system, it would be relatively easy to add/modify object definition mechanisms to automatically support e.g. monitor-like mutual exclusion, but so far this has not been a major concern in designing Kevo.

Reflection. Kevo incorporates a number of features which allow the programmer to reason about the system, and to change its behavior while it is running. The use of such introspective facilities is typically referred to as *reflection*. Among Kevo's reflective facilities, the most useful ones are:

- runtime decompilation,
- execution tracing, and
- runtime debugging of individual tasks.

Unlike most other systems with reflective facilities, Kevo does not keep source code hanging around at runtime. Instead, Kevo takes advantage of the easy decompilability of its object code format (indirect threaded code), and is capable of reproducing a textual description from almost every definition in the system. Using this feature, the user can decompile individual properties of objects, or produce mirror images of whole objects at runtime. The decompilation facilities are utilized also in *execution tracing*. When execution tracing mode is on, the system will run hundreds of times slower than in the normal execution mode, allowing the user to see precisely how the system works. Every single executed instruction and task switch will be reported to a special console window along with the contents of execution stacks. Kevo also supports dynamic debugging of concurrent tasks by allowing the user to set and remove breakpoints to particular places in code. These breakpoints will be applied only to user-specified tasks in the system, while the other tasks will continue operating normally.

In addition to the above mentioned reflective facilities, the whole Kevo kernel is reflective in the sense that the command shell, threaded code compiler, control structures, and object definition primitives have been written using the Kevo kernel itself. Since everything in Kevo is dynamically redefinable, this means that a proficient user can tailor the system to his needs even when the system is running. This characteristics is familiar from other

reflective object-oriented systems such as Smalltalk-80 [GoR83, GoR89] or CLOS [BDG88, Kee89], but owing to the self-sufficiency of objects in Kevo no complicated metaobject hierarchies are needed for achieving this effect.

5. Status and future directions

Two versions of the Kevo system exist at the moment: one for Macintosh and one for Unix. Due to limited time at our disposal, the attention is currently however focused mainly on the Macintosh implementation. As a result, the Unix version lacks any kind of graphical user interface, but we are hoping to be able to extend the system to support X Windows later on.

The Macintosh version of Kevo has a multiwindow interface, which allows the user to maneuver in the object hierarchy, govern the execution of tasks, change the general multitasking settings, and control the debugging facilities directly from the menus. Definition of objects is still essentially text-based, but we are currently in the process of building an object browser which will support object definition directly from the user interface. In general, since the concatenation approach is very suitable for programming by example, we are moving away from text-based programs towards more tangible and visual forms of programming.

The current Kevo implementation still has some limitations. Some of the module operations have been implemented using shortcuts, and as a result objects are not conceptually truly individuals e.g. when their operations are being redefined. Although Kevo has dynamic memory management, automatic garbage collection is also still missing, and needs to be implemented later. One of the reasons why the implementation of garbage collection has been postponed is the fact that there are still some more interesting issues, such as how make the Kevo system to support distributed programming, which might cause changes in the underlying object storage model. Intuitively, concatenated objects seem like an ideal platform for distributed programming owing to their self-contained, self-descriptive nature.

In general, the "ultimate" goal in designing Kevo is to create an interactive, portable, reflective, multitasking object-oriented system with a very tangible and elegant object model, clear system organization, and with the support for distributed and highly visual programming. At the time of this writing, all the goals except for the last two have been mostly fulfilled.

6. Related work

The suggested concatenation model bears much similarity to the model proposed by Alan Borning a few years ago [Bor86] (see also [Coo87]). In that model, only prototypical are supported, and new objects are created by copying existing objects similarly as in the model proposed here. Propagation of dynamic changes is achieved by defining *constraints*: relations that must hold between objects. Two separate creation operations, *copy* and

descendant, are used to create new objects which are either fully individual or incremental modifications of existing ones. In [Bor86], the model was compared to conventional class inheritance, and some suggestions on how to implement the model in Smalltalk were given. Otherwise the model was however mainly a small-scale experiment whose development was not taken any further.

Several papers on prototype-based programming have been written in the past years [Bor81, Wha83, LaP85, Bor86, Lie86, LTP86, Ste87, UnS87, SLU88, AkT88, Mer88, Sci89, LaL89], and these papers have greatly influenced this work. The achievements by the Self Group have been particularly inspiring; a comprehensive collection of their papers is available in electronic format via anonymous FTP from '@otis.stanford.edu'. Although the model suggested in this paper can be seen as an alternative to delegation-based systems such as Self, several aspects of the Self system, such as the map mechanism, have been helpful in making the concatenation model more practical. The virtual copying techniques by Mittal, Bobrow, and Kahn [MBK86] have also guided us in this process.

On the theoretical side, the work by Cook, Palsberg, and Bracha [Coo89, CoP89, BrC90] has taught us to understand the intricacies of inheritance better. In particular, although the Kevo system has actually been around in different forms for several years (starting from [Tai89a, Tai89b]), Gilad Bracha's recent work on module operations [BrL92, Bra92] has considerably helped shape Kevo towards its current form.

Recently there have appeared some interesting new object-oriented languages whose object model seems to bear some similarity to Kevo. Two such systems are *Ellie* [And91, And92] and *Omega* [Bla91]. Both of these systems are based on prototypical objects, and abandon delegation as the basic incremental modification mechanism. Ellie – a fine-grained parallel language for distributed computers – gains the effect of inheritance by a mechanism called *interface inclusion* [And92]. This mechanism, which might also be referred to as *selective inheritance* in terms of [Coo89, Tai91b], allows the programmer to explicitly include (select) properties of existing objects to the interfaces of new objects. The same kind of a mechanism, which can be seen as an intermediate form between pure delegation and concatenation, has also been used earlier in the language *Sina* [AkT88, Aks89]. Sina is however oriented more towards traditional class-(or type-)based than prototype-based programming.

The object model of *Omega* [Bla91] comes even closer to concatenation. In that system, objects are defined as linear collections of properties, and new objects are instantiated by copying existing objects. Omega however differs from Kevo in the sense that it does not allow modification of objects on an individual basis; rather, modifications to objects will automatically have effect on (≈ propagated to) all the objects of the same type. As a result, inheritance in Omega is a separate, orthogonal operation with respect to object copying, whereas in Kevo inheritance is achieved as the combined result of copying and subsequent modification. This difference arises probably from the fact that Omega is designed to support static type checking, and the individual modifiability of objects would have made the design of Omega's type system more difficult.

Note that although Ellie, Omega, and Kevo all are prototype-based object-oriented systems, none of them features delegation as an explicit language feature. This observation clearly shows that it is not generally reasonable to equate prototype-based programming with the notion of delegation, although this has been a common practice in the recent literature. We hope that this paper, for its part, has been able to shed light on this issue.

7. Conclusions

Kevo is a portable, interactive, reflective, multitasking, prototype-based object-oriented language whose main contribution is its unique object model. As opposed to many previous prototype-based systems, which use *delegation* to achieve incremental modification (inheritance), Kevo is based on *concatenation*: linear composition of object interfaces. This composition is performed by utilizing object copying and the individual modifiability of objects in the presence of the late-bound self-reference. In the typical case, inheritance is achieved simply by copying an existing object, and then adding new properties to the copy so as to differentiate the copy from its "parent". Further instances of objects can be created simply by copying existing ones.

The concatenation model eliminates the need for delegation and parent slots from prototype-based programming, and results in a very simple and concrete model of object-oriented programming in which sharing plays a considerably less crucial rôle than in delegation. This reduces implementation dependencies between objects, and makes objects more truly "prototypical". To gain systematic control over object definition and modification, Kevo features special module operations. These module operations capture all the traditional uses of inheritance, and allow individual objects as well as groups of objects to be tailored flexibly at runtime. In order to provide crisp system organization, Kevo features two orthogonal object hierarchies: organization and behavior hierarchies. Organization (≈ part-whole) hierarchy serves as a file system-like skeleton for system organization and is managed by the user. Behavior (clone family) hierachy captures the commonalities between objects and is managed automatically by the system. Together these hierarchies make it possible to clearly distinguish between objects e.g. belonging to different applications, without losing the ability to easily find suitable candidates for reuse when needed.

To conclude, the linear, self-contained nature of objects in Kevo provides a concrete and intuitive metaphor for programming: an object is simply a self-sufficient collection of its properties, nothing more and nothing less. Each object describes its own behavior, and the only basic relationships between objects are the component relationships between objects and their variables and operations. This approach implicitly hides most of the implementation decisions pertaining to sharing/duplication, and thus lends itself to various optimizations as well as concurrent and distributed programming. In general, Kevo shows that the essence of object-oriented programming can be captured using only a small number of tangible and intuitive ideas.

Acknowledgements

While designing and building the Kevo system, I have benefited from discussions with many researchers, including Markku Sakkinen (University of Jyväskylä, Finland), Kai Koskimies (University of Tampere, Finland), Peter Grogono (Concordia University, Montréal, Canada), Gilad Bracha (University of Utah, Salt Lake City, UT), David Ungar and Randall B. Smith (Sun Microsystems Laboratories, Mountain View, CA), Karl Lieberherr and Ian Holland (Northeastern University, Boston, MA), Craig Chambers (University of Washington, Seattle, WA), Bjorn Freeman-Benson and Jan Vitek (University of Victoria, B.C., Canada). This research was supported by the Academy of Finland under projects 1061120 and 8001552.

References

AkT88 Aksit,M., Tripathi,A., Data abstraction mechanisms in Sina/st. In [OOP88] pp.267-275

Ame87 America,P., Inheritance and subtyping in a parallel object-oriented language. In Bézivin,J., Hullot, J-M., Cointe,P., Lieberman,H. (eds): ECOOP'87 European Conference on Object-oriented Programming (Paris, France, June 15-17), Lecture Notes in Computer Science 276, Springer-Verlag, 1987, pp.234-242

Ame90 America,P., Designing an object-oriented programming language with behavioural subtyping. In [FOO91] pp.60-90

And91 Andersen,B., Ellie language definition report. Partial Ph.D. dissertation, University of Copenhagen, Denmark, DIKU Report 91/3, June 1991

And92 Andersen,B., Ellie: a general, fine-grained, first-class, object-based language. Journal of Object-Oriented Programming vol 5, nr 2 (May) 1992, pp.35-42

Bla91 Blaschek,G., Type-safe OOP with prototypes: the concepts of Omega. Structured Programming vol 12, nr 12 (Dec) 1991, pp.1-9

BDG88 Bobrow,D.G., DeMichiel,L.G., Gabriel,R.P., Keene,S.E., Kiczales,G., Moon,D.A., Common Lisp Object System specification X3J13 document 88-002R. ACM SIGPLAN Notices vol 23 special issue, September 1988

BKK86 Bobrow,D.G., Kahn,K., Kiczales,G., Masinter,L., Stefik,M., Zdybel,F., CommonLoops: merging Lisp and object-oriented programming. In [OOP86] pp.17-29

Bor81 Borning,A.H., The programming language aspects of ThingLab, a constraint-oriented simulation laboratory. ACM Transactions on Programming Languages and Systems vol 3, nr 4 (Oct) 1981, pp.353-387

Bor86 Borning,A.H., Classes versus prototypes in object-oriented languages. In Proceedings of ACM/IEEE Fall Joint Computer Conference, November 1986, pp.36-40

BoO87 Borning,A., O'Shea,T., Deltatalk: an empirically and aesthetically motivated simplification of the Smalltalk-80 language. In Bézivin,J., Hullot,J-M., Cointe,P., Lieberman,H. (eds): ECOOP'87 European Conference on Object-oriented Programming (Paris, France, June 15-17), Lecture Notes in Computer Science 276, Springer-Verlag, 1987, pp.1-10

Bra92 Bracha,G., The programming language Jigsaw: mixins, modularity and multiple inheritance. Ph.D. thesis, University of Utah, March 1992

BrC90 Bracha,G., Cook,W., Mixin-based inheritance. In Meyrowitz,N. (ed): OOPSLA/ECOOP'90 Conference Proceedings (Ottawa, Canada, October 21-25), ACM SIGPLAN Notices vol 25, nr 10 (Oct) 1990, pp.303-311

BrL92 Bracha,G., Lindstrom,G., Modularity meets inheritance. In Proceedings of the 1992 International Conference on Computer Languages (Oakland, California, April 20-23), IEEE Computer Society Press, 1992, pp.282-290

ChU91 Chambers,C., Ungar,D., Making pure object-oriented languages practical. In Paepcke,A. (ed): OOPSLA'91 Conference Proceedings (Phoenix, Arizona, October 6-11), ACM SIGPLAN Notices vol 26, nr 11 (Nov) 1991, pp.1-15

CUC91 Chambers,C., Ungar,D., Chang,B-W., Hölzle,U., Parents are shared parts of objects: inheritance and encapsulation in Self. Lisp and Symbolic Computation vol 4, nr 3 (Jun) 1991

CUL89 Chambers,C., Ungar,D., Lee,E.H-S., An efficient implementation of Self, a dynamically-typed object-oriented language based on prototypes. In [OOP89] pp.49-70

Coo87 Cook,S. (ed), OOPSLA'87 panel: Varieties of inheritance. In Power,L., Weiss,Z. (ed): Addendum to the Proceedings of OOPSLA'87, ACM SIGPLAN Notices vol 23, nr 5 (May) 1988, pp.35-40

Coo89 Cook,W.R., A denotational semantics of inheritance. Ph.D. thesis, Brown University technical report CS-89-33, May 1989

CHC90 Cook,W.R., Hill,W.L., Canning,P.S., Inheritance is not subtyping. In Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (San Francisco, California, January 17-19), ACM Press, 1990, pp.125-135

CoP89 Cook,W.R., Palsberg,J., A denotational semantics of inheritance and its correctness. In [OOP89] pp.433-443

DDH72 Dahl,O-J., Dijkstra,E.W., Hoare,C.A.R., Structured programming. Academic Press, London, 1972

DMN68 Dahl,O-J., Myhrhaug,B., Nygaard,K., SIMULA 67 common base language. Norwegian computing center, Oslo, May 1968

Dew75 Dewar,R.B.K., Indirect threaded code. Communications of the ACM vol 18, nr 6 (Jun) 1975, pp.330-331

Duf84 Duff,C., Neon – extending Forth to new dimensions. In Proceedings of the 1984 FORML Conference (Pacific Grove, California, November 23-25), Forth Interest Group, 1984

Duf86 Duff,C., Designing an efficient language. BYTE vol 11, nr 8 (Aug) 1986, pp.211-224

ECO88 Gjessing,S., Nygaard,K. (eds), ECOOP'88 European Conference on Object-Oriented Programming (Oslo, Norway, August 15-17). Lecture Notes in Computer Science 276, Springer-Verlag, 1988

ECO89 Cook,S. (ed), ECOOP'89: Proceedings of the Third European Conference on Object-Oriented Programming (Nottingham, England, July 10-14). The British Computer Society workshop series, Cambridge University Press, 1989

ElS90 Ellis,M.A., Stroustrup,B., The annotated C++ reference manual. Addison-Wesley, 1990

FOO91 De Bakker,J.W., De Roever,W.P., Rozenberg,G. (eds), Foundations of object-oriented languages. REX School/Workshop (Noordwijkerhout, The Netherlands, May 28–June 1, 1990) Proceedings, Lecture Notes in Computer Science 489, Springer-Verlag, 1991

Fra71 Fraser,A.G., On the meaning of names in programming systems. Communications of the ACM vol 14, nr 6 (Jun) 1971, pp.409-416

GoR83 Goldberg,A., Robson,D., Smalltalk-80: the language and its implementation. Addison-Wesley, 1983

GoR89 Goldberg,A., Robson,D., Smalltalk-80: the language. Addison-Wesley, 1989

Hal88 Halbert,D., Types vs. prototypes. In [OOP88] p.362

HaP91 Harper,R., Pierce,B., A record calculus based on symmetric concatenation. In Proceedings of the

Eighteenth Annual ACM Symposium on Principles of Programming Languages, ACM Press, 1991, pp.131-142

Kee89 Keene,S.E., Object-oriented programming is Common Lisp: a programmer's guide to CLOS. Addison-Wesley, 1989

KMM83 Kristensen,B.B., Madsen,O.L., Møller-Pedersen,B., Nygaard,K., Abstraction mechanisms in the Beta programming language. In Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages (Austin, Texas, January 24-26), ACM Press, 1983, pp.285-298

LaP85 LaLonde,W.R., Pugh,J.R., Features of fifth generation languages: a panoramic view. Carleton University computer science report SCS-TR-70, March 1985

LTP86 LaLonde,W.R., Thomas,D.A., Pugh,J.R., An exemplar based Smalltalk. In [OOP86] pp.322-330

LaL89 LaLonde,W.R., Designing families of data types using exemplars. ACM Transactions on Programming Languages and Systems vol 11, nr 2 (Apr) 1989, pp.212-248

Lie86 Lieberman,H., Using prototypical objects to implement shared behavior in object-oriented systems. In [OOP86] pp.214-223

Mac82 MacLennan,B.J., Values and objects in programming languages. ACM SIGPLAN Notices vol 17, nr 12 (Dec) 1982, pp.70-79

MaM88 Madsen,O.L., Møller-Pedersen,B., What object-oriented programming may be and what it does not have to be? In [ECO88] pp.1-20

Mag91 Magnusson,B., Implementation of inheritance in Simula. Personal communication, 1991

Mer88 Mercado,A.Jr., Hybrid: implementing classes with prototypes. Brown University technical report, July 1988

Mey90 Meyers,S., Working with object-oriented programs: the view from the trenches in not always pretty. In Proceedings of the SOOPPA Symposium on Object-Oriented Programming Emphasizing Practical Applications (Poughkeepsie, NY, September 14-15), 1990

MBK86 Mittal,S., Bobrow,D.G., Kahn,K.M., Virtual copies – at the boundary between classes and instances. In [OOP86] pp.159-166

Moo86 Moon,D.A., Object-oriented programming with Flavors. In [OOP86] pp.1-8

NyD78 Nygaard,K., Dahl,O-J., The development of the Simula languages. In Wexelblat,R.L. (ed): ACM SIGPLAN History of Programming Languages Conference (Los Angeles, California, June 1-3), ACM SIGPLAN Notices vol 13, nr 8 (Aug) 1978, pp.245-272

OOP86 Meyrowitz,N. (ed), OOPSLA'86 Conference Proceedings (Portland, Oregon, September 26–October 2). ACM SIGPLAN Notices vol 21, nr 11 (Nov) 1986

OOP87 Meyrowitz,N. (ed), OOPSLA'87 Conference Proceedings (Orlando, Florida, October 4-8). ACM SIGPLAN Notices vol 22, nr 12 (Dec) 1987

OOP88 Meyrowitz,N. (ed), OOPSLA'88 Conference Proceedings (San Diego, California, September 25-30). ACM SIGPLAN Notices vol 23, nr 11 (Nov) 1988

OOP89 Meyrowitz,N. (ed), OOPSLA'89 Conference Proceedings (New Orleans, Louisiana, October 1-6). ACM SIGPLAN Notices vol 24, nr 10 (Oct) 1989

OXF33 Oxford English Dictionary VIII. Oxford University Press, 1933

PaS91 Palsberg,J., Schwartzbach,M.I., Static typing for object-oriented programming. Aarhus University technical report DAIMI PB-355, Denmark, June 1991

RaL89 Raj,R.K., Levy,H.M., A compositional model for software reuse. In [ECO89] pp.3-24

Sch86 Schmucker,K: Object-oriented languages for the Macintosh. BYTE vol 11, nr 8 (Aug) 1986, pp.177-185

Sci89 Sciore,E., Object specialization. ACM Transactions on Information Systems vol 7, nr 2 (Apr) 1989, pp.103-122

Sny87 Snyder,A., Inheritance and the development of encapsulated software components. In [RDO87] pp.165-188

Ste87 Stein,L.A., Delegation is inheritance. In [OOP87] pp.138-146

SLU88 Stein,L.A., Lieberman,H., Ungar,D., A shared view of sharing: the treaty of Orlando. In Kim,W., Lochowsky,F. (ed): Object-oriented concepts, applications and databases, Addison-Wesley, 1988, pp.31-48.

TGP89 Taenzer,D., Ganti,M., Podar,S., Problems in object-oriented software reuse. In [ECO89] pp.25-38

Tai89a Taivalsaari,A., Inheritance in object-oriented programming: class and prototype approaches. Master's thesis, Computer Science Publications TU-6, University of Jyväskylä, Finland, June 1989 (in Finnish)

Tai89b Taivalsaari, A., Cool – unifying class and prototype inheritance. In Proceedings of the 11th FORML Conference (Pacific Grove, California, November 24-26), Forth Interest Group, 1989, pp.240-248

Tai91a Taivalsaari,A., Cloning is inheritance. Computer science report WP-18, University of Jyväskylä, Finland, February 1991, 17 pages

Tai91b Taivalsaari,A., Towards a taxonomy of inheritance mechanisms in object-oriented programming. A thesis for the degree of licentiate. Published as Computer science report WP-19, University of Jyväskylä, Finland, September 1991, 178 pages

Tai92 Taivalsaari,A., Object-oriented programming with modes. To appear in Journal of Object-Oriented Programming

UCC91 Ungar,D., Chambers,C., Chang,B-W., Hölzle,U., Organizing programs without classes. Lisp and Symbolic Computation vol 4, nr 3 (Jun) 1991

UnS87 Ungar,D., Smith,R.B., Self: the power of simplicity. In [OOP87] pp.227-241

Wan89 Wand,M., Type inference for record concatenation and multiple inheritance. In Proceedings of the IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, 1989, pp.92-97

Weg87 Wegner,P., The object-oriented classification paradigm. In Shriver,B., Wegner,P. (ed): Research directions in object-oriented programming, The MIT Press, 1987, pp.479-560

Weg90 Wegner,P., Concepts and paradigms of object-oriented programming. ACM OOPS Messenger vol 1, nr 1 (Aug) 1990, pp.7-87

WeZ88 Wegner,P., Zdonik,S.B., Inheritance as an incremental modification mechanism or what Like is and isn't like. In [ECO88] pp.55-77

Wha83 Wharton,R.M., A note on types and prototypes. ACM SIGPLAN Notices vol 18, nr 12 (Dec) 1983, pp.122-126

Whi90 The Whitewater Group, Actor User's Manual volume 1. The Whitewater Group, 1990

Technical Report, LACIR 92-02

Department of Computer Science

DCS-197-IR

University of Victoria
P.O. Box 3055
Victoria, BC. Canada
V8W 3P6

Phone: 604.721.7297
Fax: 604.721.7292
EMail: lacir@csr.uvic.ca